# GPU Programming

# Mutual Exclusion

Christian Lessig

# Parallelism and Concurrency

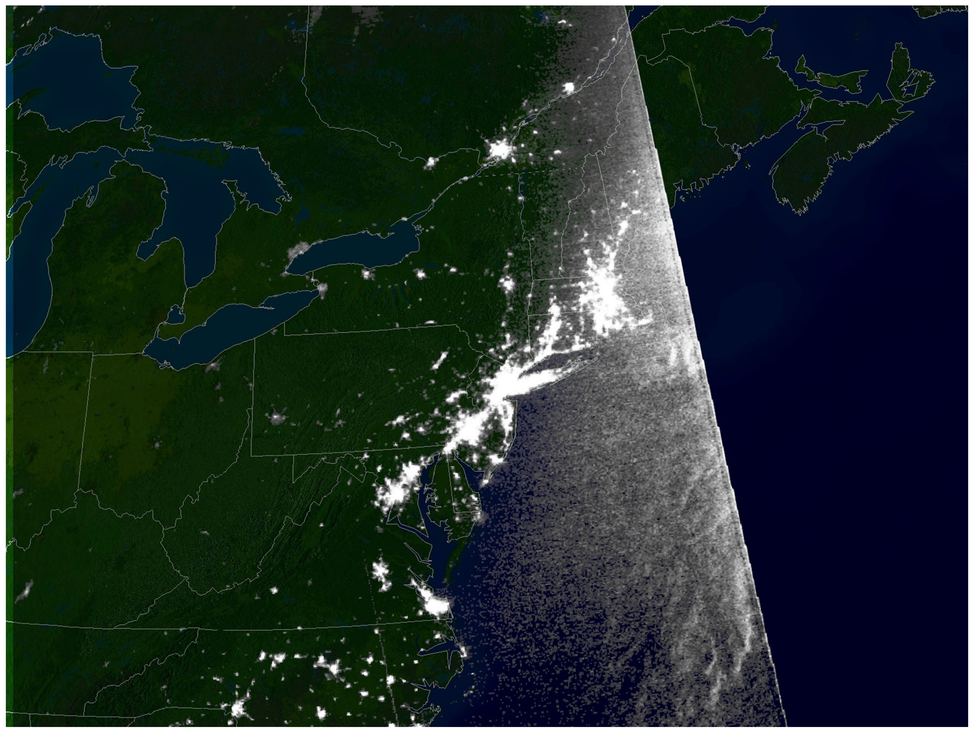
Parallelism: use multiple resources simultaneously to obtain an answer more efficiently

# Parallelism and Concurrency

Parallelism: use multiple resources simultaneously to obtain an answer more efficiently

Concurrency: correct and efficient use of simultaneously used resources

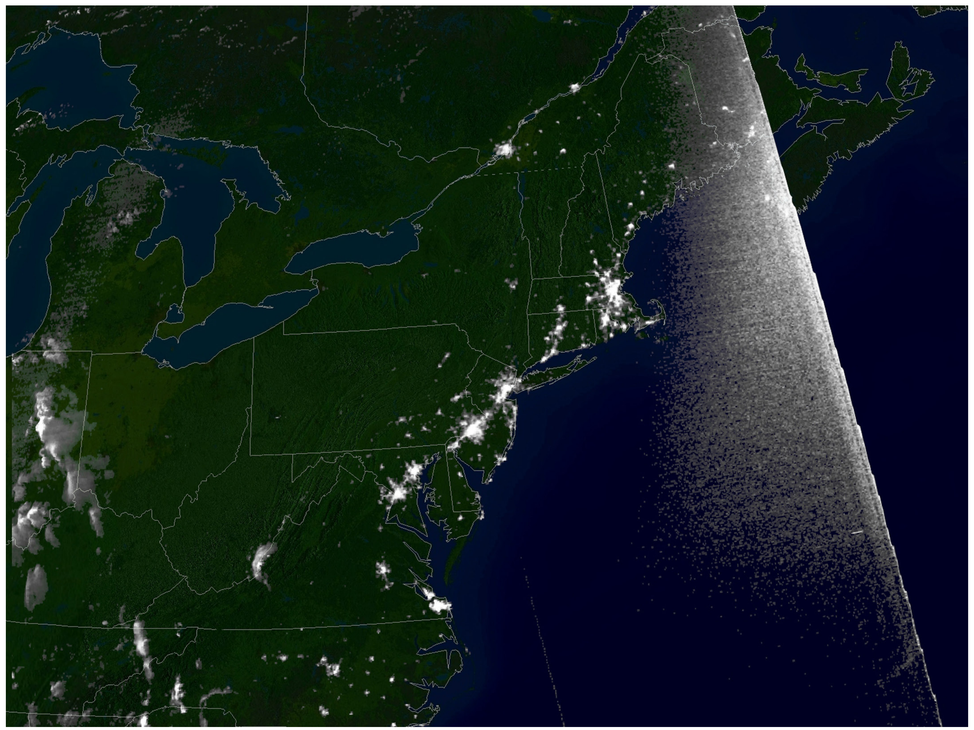
# Concurrency



http://www.noaanews.noaa.gov/stories/s2015.htm

North-Eastern USA and Canada

# Concurrency



http://www.noaanews.noaa.gov/stories/s2015.htm

North-Eastern USA and Canada

Blackout in 2003

# Parallelism

Compute in parallel:

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

# Parallelism

Compute in parallel:

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

$$= \sum_{i=1}^{n/2} a_i \, b_i + \sum_{i=n/2}^{n} a_i \, b_i$$

# Parallelism

Compute in parallel:

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

$$= \boxed{\sum_{i=1}^{n/2} a_i \, b_i} + \boxed{\sum_{i=n/2}^{n} a_i \, b_i}$$

compute independently and then combine

# Concurrency

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i =1 : n/2
 sum += a[i] * b[i]
```

```
// T2
for i =n/2 : n
 sum += a[i] * b[i]
```

# Concurrency

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i =1 : n/2
 sum += a[i] * b[i]
```

```
// T2
for i =n/2 : n
 sum += a[i] * b[i]
```

race condition

# Concurrency

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i =1 : n/2
 sum += a[i] * b[i]
```

```
// T2
for i =n/2 : n
 sum += a[i] * b[i]
```

race condition

- non-deterministic behaviour
- program is incorrect

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program
- Requirements:
  - Mutual exclusion
  - No starvation: every thread / process gets access
  - No deadlock: program terminates

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program
- Desirables:
  - Minimal serialization
  - Minimal overhead
  - Minimal overall execution time

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program
  - (avoid dependencies)

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program
  - (avoid dependencies)
  - Atomic

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program
  - (avoid dependencies)
  - Atomic
  - Mutex

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program
  - (avoid dependencies)
  - Atomic
  - Mutex
  - Semaphore

# Mutual exclusion

- Mechanism to avoid race condition and ensure correctness of program
  - (avoid dependencies)
  - Atomic
  - Mutex
  - Semaphore
  - Barrier

# Avoid dependencies

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i =1 : n/2
 sum += a[i] * b[i]
```

```
// T2
for i =n/2 : n
 sum += a[i] * b[i]
```

# Avoid dependencies

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
s1 = 0, s2 = 0
```

```
// T1
for i = 1 : n/2
  s1 += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  s2 += a[i] * b[i]
```

# Avoid dependencies

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i = 1 : n/2
  s1 += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  s2 += a[i] * b[i]
```

```
// serial sum
sum = s1 + s2
```

# Avoid dependencies

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i = 1 : n/2
  s1 += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  s2 += a[i] * b[i]
```

```
// serial sum
sum = s1 + s2
```

synchronization required

# Avoid dependencies

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i = 1 : n/2
  s1 += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  s2 += a[i] * b[i]
```

```
for( thread : threads) { thread.join(); }
```

```
// serial sum
sum = s1 + s2
```

# Barrier

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i = 1 : n/2
  s1 += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  s2 += a[i] * b[i]
```

```
for( thread : threads) { thread.join(); }
```

```
// serial sum
sum = s1 + s2
```

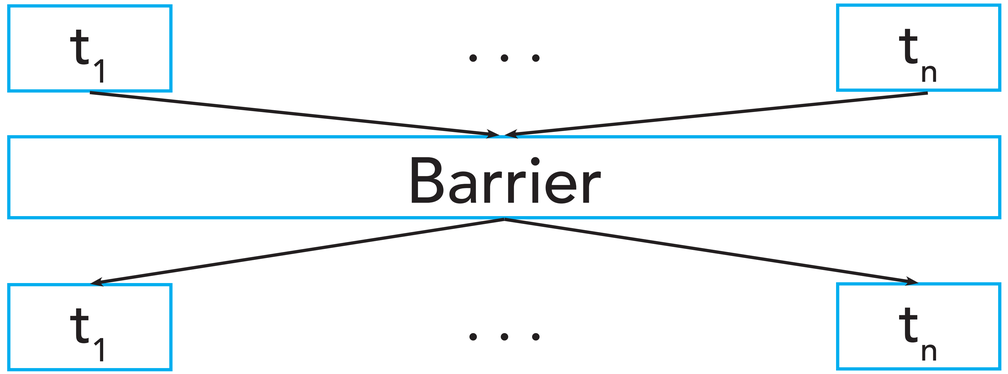
realizes barrier

# Barrier

- Primitive that ensures that no thread will proceed past barrier until all reached it

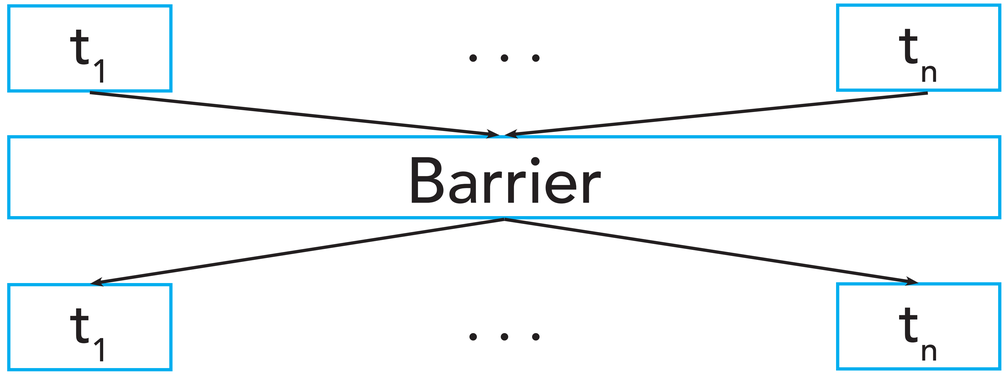
# Barrier

- Primitive that ensures that no thread will proceed past barrier until all reached it



# Barrier

- Primitive that ensures that no thread will proceed past barrier until all reached it




- Useful when intermediate result has to be finished before proceeding

# Atomic

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
sum = 0
```

```
// T1
for i = 1 : n/2
  s1 += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  s2 += a[i] * b[i]
```

# Atomic

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
std::atomic<int> sum( 0)
```

```
// T1
for i = 1 : n/2
  sum += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  sum += a[i] * b[i]
```

# Atomic

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
std::atomic<int> sum( 0)
```

```
// T1
for i = 1 : n/2
  sum += a[i] * b[i]
```

```
// T2
for i = n/2 : n
  sum += a[i] * b[i]
```



has to be POD

# Mutex

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
int sum = 0
std::mutex mutex_sum
```

```
// T1
for i = 1 : n/2

  sum += a[i] * b[i]
```

```
// T2
for i = n/2 : n

  sum += a[i] * b[i]
```

# Mutex

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
int sum = 0
std::mutex mutex_sum
```

```
// T1
for i = 1 : n/2
  mutex_sum.lock()
  sum += a[i] * b[i]
  mutex_sum.unlock()
```

```
// T2
for i = n/2 : n
  mutex_sum.lock()
  sum += a[i] * b[i]
  mutex_sum.unlock()
```

# Mutex

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \, b_i$$

```
// Initialize
int sum = 0
std::mutex mutex_sum
```

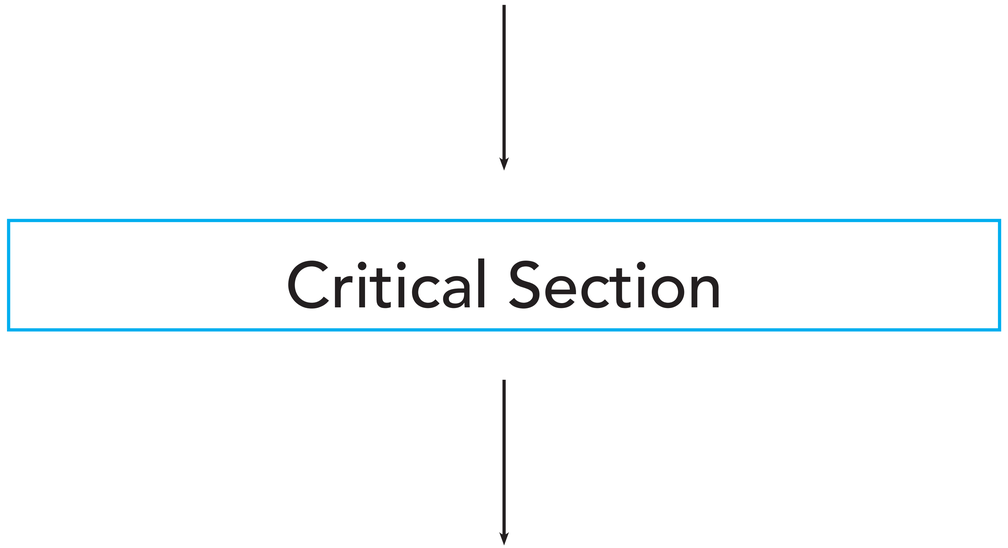
```
// T1
for i = 1 : n/2
  mutex_sum.lock()
  sum += a[i] * b[i]
  mutex_sum.unlock()
```

```
// T2
for i = n/2 : n
  mutex_sum.lock()
  sum += a[i] * b[i]
  mutex_sum.unlock()
```

can be arbitrarily complex

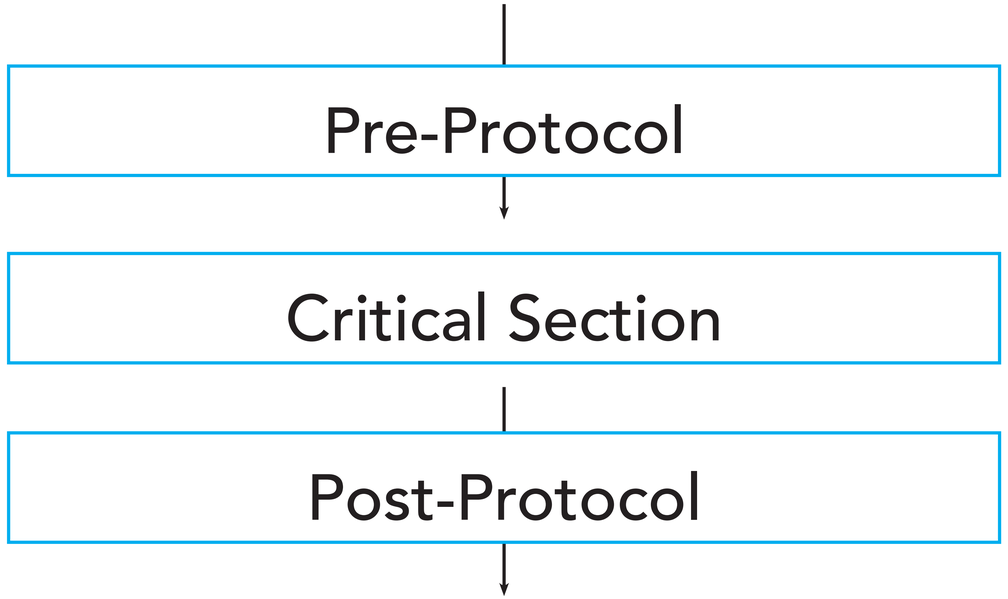
# Mutex

- Principle:



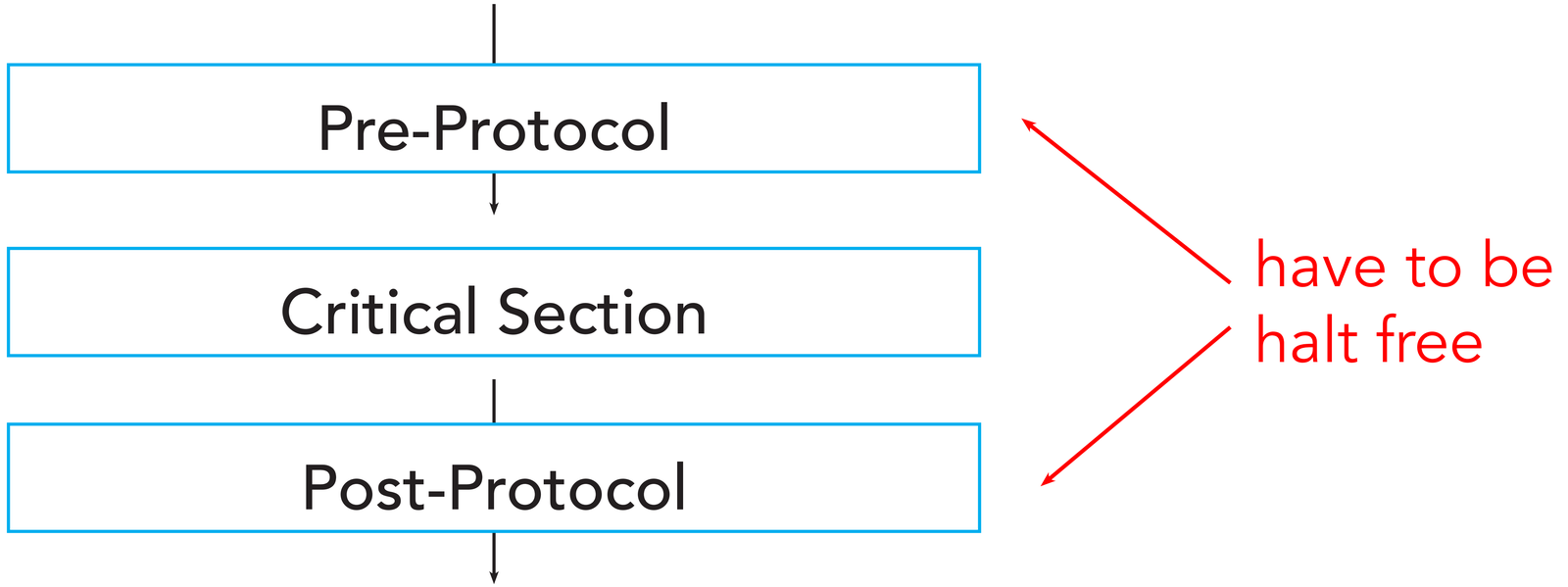
# Mutex

- Principle:



# Mutex

- Principle:



# Mutex versus Atomic

- Mutex: can enclose arbitrary code, but significant overhead
- Atomic: only available for plain old datatypes, but less overhead than mutex

# Software mutex

# Software mutex

thread 1

```
. . .

. . .   critical section

. . .
```

thread 2

```
. . .

. . .   critical section

. . .
```

# Software mutex

```
bool in_critical[2] = {false, false};
```

```
...

...  critical section

...
```

```
...

...  critical section

...
```

# Software mutex

```
bool in_critical[2] = {false, false};
```

# Software mutex

```
bool in_critical[2] = {false, false};
```

```
...
while in_critical[1] { }
in_critical[0] = true;

...

in_critical[0] = false;
```

```
...
while in_critical[0] { }
in_critical[1] = true;

...

wants_entry[1] = false;
```

# Software mutex

```
bool in_critical[2] = {false, false};
```

```
...
while in_critical[1] { }
in_critical[0] = true;

...

in_critical[0] = false;
```

```
...
while in_critical[0] { }
in_critical[1] = true;

...

wants_entry[1] = false;
```

**no mutual exclusion**

# Software mutex

```
bool wants_entry[2] = {false, false};
```

# Software mutex

```
bool wants_entry[2] = {false, false};
```

```
wants_entry[0] = true;
while wants_entry[1] { }


...
wants_entry[0] = false;
```

```
wants_entry[1] = true;
while wants_entry[0] { }


...
wants_entry[1] = false;
```

# Software mutex

```
bool wants_entry[2] = {false, false};
```

```
wants_entry[0] = true;
while wants_entry[1] { }

...
wants_entry[0] = false;
```

```
wants_entry[1] = true;
while wants_entry[0] { }

...
wants_entry[1] = false;
```

**dead lock**

# Software mutex

```
bool wants_entry[2] = {false, false};
```

```
wants_entry[0] = true;
while wants_entry[1] { }

...
wants_entry[0] = false;
```

```
wants_entry[1] = true;
while wants_entry[0] { }

...
wants_entry[1] = false;
```

# Software mutex

```
bool wants_entry[2] = {false, false};
```

```
wants_entry[0] = true;
while wants_entry[1] {
  wants_entry[0] = false;
  // wait a bit
  wants_entry[0] = true;
}

...
wants_entry[0] = false;
```

```
wants_entry[1] = true;
while wants_entry[0] {
  wants_entry[1] = false;
  // wait a bit
  wants_entry[1] = true;
}

...
wants_entry[1] = false;
```

# Software mutex

```
bool wants_entry[2] = {false, false};
```

```
wants_entry[0] = true;
while wants_entry[1] {
  wants_entry[0] = false;
  // wait a bit
  wants_entry[0] = true;
}

...
wants_entry[0] = false;
```

```
wants_entry[1] = true;
while wants_entry[0] {
  wants_entry[1] = false;
  // wait a bit
  wants_entry[1] = true;
}

...
wants_entry[1] = false;
```

**starvation**

# Software mutex

```
bool wants_entry[2] = {false, false};
unsigned int turn = 0;
```

# Software mutex

```
bool wants_entry[2] = {false, false};
unsigned int turn = 0;
```

```
wants_entry[0] = true;
while wants_entry[1] {
 if (0 != turn) {
  wants_entry[0] = false;
  while (0 != turn) { }
  wants_entry[0] = true;
 }
}
...
```

```
wants_entry[1] = true;
while wants_entry[0] {
 if (1 != turn) {
  wants_entry[1] = false;
  while (1 != turn) { }
  wants_entry[1] = true;
 }
}
...
```

# Software mutex

```
wants_entry[0] = true;
while wants_entry[1] {
 if (0 != turn) {
  wants_entry[0] = false;
  while (0 != turn) { }
  wants_entry[0] = true;
 }
}


...
turn = 1;
wants_entry[0] = false;
```

```
wants_entry[1] = true;
while wants_entry[0] {
 if (1 != turn) {
  wants_entry[1] = false;
  while (1 != turn) { }
  wants_entry[0] = true;
 }
}


...
turn = 0;
wants_entry[1] = false;
```

# Dekker's algorithm

```
wants_entry[0] = true;
while wants_entry[1] {
 if (0 != turn) {
  wants_entry[0] = false;
  while (0 != turn) { }
  wants_entry[0] = true;
 }
}
```

critical section

```
...
turn = 1;
wants_entry[0] = false;
```

```
wants_entry[1] = true;
while wants_entry[0] {
 if (1 != turn) {
  wants_entry[1] = false;
  while (1 != turn) { }
  wants_entry[1] = true;
 }
}
```

critical section

```
...
turn = 0;
wants_entry[1] = false;
```

# Dekker's algorithm

- Requirements:
  - Mutual exclusion
  - No starvation
  - No deadlock
- Desirables:
  - Minimal serialization
  - Minimal overhead

# Dekker's algorithm

- Requirements:
  - Mutual exclusion
  - No starvation
  - No deadlock
- Desirables:
  - Minimal serialization
  - Minimal overhead

Satisfied

# Dekker's algorithm

- Requirements:
  - Mutual exclusion
  - No starvation
  - No deadlock
- Desirables:
  - Minimal serialization
  - Minimal overhead

Satisfied

Good

# Dekker's algorithm

- Requirements:
  - Mutual exclusion
  - No starvation
  - No deadlock
- Desirables:
  - Minimal serialization
  - Minimal overhead

Satisfied

Good

Active waiting

# Further reading


- T. Rauber and G. Rünger, *Parallel Programming*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.
- N. Matloff, Programming on Parallel Machines. 2016.
- P. E. McKenney, Is Parallel Programming Hard, And, If So, What Can You Do About It? 2016.
- M. Raynal, Concurrent Programming: Algorithms, Principles, and Foundations, Springer, 2013.